摘　要

[**摘　要**] 随着平板电脑和智能手机的普及，每天都会产生海量数据。为了对这些数据进行存储，独立磁盘冗余阵列（Redundant Arrays of Independent Disks，RAID）利用多块独立的磁盘，组合成一个容量巨大的磁盘组，从而利用个别磁盘提供数据所产生加成效果提升整个磁盘系统效能。在 RAID 系统中，由于磁盘数较多，磁盘故障经常发生。为避免磁盘故障对整个系统的影响，RAID 系统通过存储冗余数据，来保证数据可靠性。当磁盘发生故障时，利用冗余数据计算和恢复故障磁盘中存储的数据。同时，为了提升用户体验和避免磁盘恢复过程中更多磁盘损坏，磁盘恢复的速度要尽可能快。但是，磁盘读写性能发展缓慢，利用传统的 RAID 系统恢复海量数据时需要耗费过多时间。

本文就如何提高 RAID5 和基于 Reed-Solomon 算法的 RAID6 编码和恢复速度进行研究，提出一种应用 AVX（Advanced vector Extensions）指令，通过单线程并行优化的方法，改进原有算法内部函数运算方式。实验结果表明，本文提出的指令改进有效提高了 RAID 系统编码和解码速度。在确保准确率的同时提高速度约 230%，显著提升算法的时间性能，实现了 RAID 系统编码和解码在算法改写方向的并行优化。

[**关键词**] RAID；AVX 指令集；Reed-Solomon

Abstract

[Abstract] With the popularity of tablets and smartphones, huge amounts of data are generated every day.To store this data, Redundant Arrays of Independent Disks (RAID) use separate Disks to combine into a single large disk pack, thereby increasing the performance of the entire disk system with the addition of data provided by individual Disks.In RAID systems, disk failures occur frequently due to the high number of disks.In order to avoid the impact of disk failure on the whole system, RAID systems ensures the reliability of data by storing redundant data.When the disk fails, the redundant data is used to calculate and recover the data stored in the failed disk.Also, in order to improve the user experience and avoid more disk damage during disk recovery, disk recovery should be as fast as possible.However, disk read and write performance has been slow to develop, and it can take a lot of time to recover massive amounts of data using traditional RAID systems.

This paper studies how to improve RAID5 and RAID6 based on Reed-Solomon algorithm coding and recovery speed. By applying AVX (Advanced vector Extensions) instruction, the internal function operation of the original algorithm is improved by single thread parallel optimization.The experimental results show that the proposed instruction improvement can effectively improve the encoding and decoding speed of RAID system.To ensure the accuracy and improve the speed of about 230%, significantly improve the time performance of the algorithm, RAID system coding and decoding in the direction of algorithm rewrite parallel optimization.

[Keywords] RAID; AVX instruction set; Reed-Solomon

目录

第一章　引言

为了存储日益增长的海量数据，现代存储系统利用大规模磁盘来增加容量和读写带宽。但随着磁盘数量的增多，存储系统发生故障的概率也大大增加，因此保证數据可靠性也是存储系统的重要工作。本章首先介绍了大规模数据存储的背景，并提出使用软 RAID 的重要性；然后分别介绍了 RAID6 的几种实现方式，以及对 AVX 的简单介绍；最后介绍了本文的主要工作和整篇论文的组织结构。

1.1　选题背景与意义

衡量计算机系统的性能主要通过计算时间，而在当今这个数据爆炸式增长的时代，随着平板电脑和智能手机的普及，在日常生活中产生了越来越多的数据，因此，存储能力也成为了衡量计算机系统发展水平的重要标准。海量数据的产生为我们带来了机遇，通过对数据的检索、分析、处理可以从中得到有用信息，从而改善人们的生活。但如此多的数据对于存储系统是一个很大的挑战，如何存储数据、快速读写数据、以及确保数据可靠性都是需要处理的问题。其中可靠性无疑是最重要的问题，但由于数据爆炸式增长导致存储系统的规模和复杂性持续上升，系统的可靠性也在逐步降低。因此，如何确保数据存储的可靠性是当前需要处理的主要问题。

在实际的应用过程中，磁盘可能由于多种原因造成损坏，例如磁头组件损坏、控制电路损坏、综合性损坏、扇区物理性损坏、磁道伺服信息出错、系统信息区出错、扇区逻辑错误等，其中软件损坏是比较容易修复的，但硬件损坏难以修复。根据卡内基梅隆大学的研究人员对各种大型生产系统中的磁盘替换率的统计，在实际应用过程中磁盘的年更换率通常达到 2%∼4%[1]。随着数据规模的日益增长，数据存储所需空间也在不断提升，也就意味着存储系统所需磁盘数量不断提升，但由于技术限制，磁盘的失效率几乎不变。这就导致了存储系统规模越庞大，发生磁盘损坏的几率也就会越高。因此，磁盘的失效发生得非常频繁，几乎是一个常态事件。

如今，随着固态硬盘（Solid State Disk，SSD）和非易失存储器（NonVolatile Memory，NVM）等高速存储设备的发展，这些存储设备逐步应用在现代存储系统中。这类高速存储设备比传统磁盘有着更好的读写性能，并且具有能耗低、非易失、存储密度高等优点。然而，这类高速存储设备有一个共同的缺陷，即读写寿命有限，可靠性相对较低，这是因为它们的基本存储单元在经历过有限次数的写操作后就会失效。因此，在实际应

用过程中，使用高速存储设备无法达到提高存储系统可靠性的目的。

由以上分析可知，在实际存储系统中，硬件故障往往是不可避免的，为了确保数据的可靠性，通常使用 RAID 技术。如果采用传统的硬 RAID 方式保证可靠性，RAID 接口带宽甚至会低于存储设备的带宽，势必会限制 SSD、NVM 等高速存储设备的性能发挥。本研究拟采用软 RAID 方式，保证参与 RAID 构建的各存储设备能够独立发挥带宽优势。然而，软 RAID 依赖主机端 CPU 实施编码计算，从而占用大量的计算资源。而且，当某些硬盘发生故障时，要恢复这些磁盘上所存储的数据，需要读取多个其它磁盘上的数据并进行解码运算。因此，使用软 RAID 会增加系统的复杂性，将计算资源转换为系统的存储效率和可靠性。为了减少计算资源的消耗，需要提升 RAID 计算速度，即可提升存储系统的数据容错机制和故障发生时的数据修复机制，对于提升存储系统的可靠性有着极其重要的科学意义。

为了提升 RAID 的计算速度，目前已提出许多研究工作，其中大部分是对 RAID 算法进行优化从而加快计算过程，或是优化 RAID 存储结构来加快计算。当前，大多数 CPU 中都配备有向量指令，而一般 RAID 计算中很少利用了该项功能。通过向量指令，可以更加有效的利用 CPU 资源，提升 RAID 的计算速度。本研究将采用 AVX 指令为 RAID 编码加速，使用单线程并行优化的方法，改进原有算法内部函数运算方式，从而充分发挥 CPU 的并行处理能力，以加快 RAID 的编码和解码速度。

1.2 国内外研究现状

对于本文重点研究的 RAID6 和 RAID1 至 RAID5 不同，RAID6 是一种规范，国内外已经开发了多种用于实现 RAID6 的技术，例如 Reed-Solomon 码 [2, 3, 4, 5, 6]、Cauchy Reed-Solomon 码、EVENODD 码 [7]、RDP 码 [8]、Liberation 码、Liber8tion 码、Blaum-Roth 码、X-Code 码、B-Code 码、P-Code 码 [9] 等。下面对其中算法进行简单介绍：

（1）Reed-Solomon 码和 Cauchy Reed-Solomon 码：RS 码（Reed-Solomon）是由 Reed 和 Solomon 提出，是一种常用的纠删码，其利用范特蒙矩阵的特性来实现纠错码的功能，理论上能容任意数量的磁盘故障。但其计算过程复杂，因此采用基于有限域的方法，其编码和解码通过伽罗瓦域 $GF(2^w)$ 实现，通过查表加快计算速度。CRS 码（Cauchy Reed-Solomon）是在 RS 码的基础上进行优化，采用 Cauchy 编码矩阵，从而更快求得逆矩阵，同时将伽罗瓦域 $GF(2^w)$ 中的的运算全部转换为 XOR 运算，从而加快计算的速度。

（2）EVENODD 码：EVENODD 码是一种阵列码，可以容忍两个磁盘同时故障，可以很好的保证数据的可靠性。EVENODD 码在原始数据的基础上增加两个校验列，水

平校验列和对角线校验列，通过 XOR 运算实现数据的编码和解码。因此，其复杂度较低，编码性能高于 RS 码。

（3）RDP 码：RDP 码是在 EVENODD 码基础上优化的结果，其在阵列的构造上优于 EVENODD 码，因此其编码和解码的复杂度更低。RDP 码编码过程是先计算水平校验列，再将水平校验列和原始数据一同作为数据计算对角线校验列，通过 XOR 运算实现数据的编码和解码。

（4）Liberation 码和 Liber8tion 码：Liberation 码是一种强系统码，其更新复杂度最低，但解码比较复杂。Liber8tion 码是 Liberation 码中的一种特例，其每个列由 8 位组成，因此其每个条块的大小为 4KB，与文件系统中扇区的大小相同，可以提升系统性能[1]。

（5）Blaum-Roth 码：Blaum-Roth 码是一种阵列码，基于多项式环构造。相比于 RS 码，Blaum-Roth 码编码和解码过程可以通过循环移位实现乘法运算，因此计算效率较高。

（6）X-Code 码：X-Code 码是一种垂直码，采用垂直异或的方法实现编码和解码，其实现过程简单，而且复杂度低，但其对码长有着严格的限制。

（7）B-Code 码和 P-Code 码：B-Code 码是一种垂直码，其复杂度低，对码长的限制不严格，但实现过程复杂。P-Code 码是 B-Code 码中码长为素数的特例。

AVX 指令集是为处理器微架构 Sandy Bridge 所研发的指令扩展集，它是对 SSE 指令集的一个扩展，使用 AVX 技术进行矩阵计算的时候将比 SSE 技术快 90%，并且新增加的指令在处理性能上比 SSE 系列更为先进。

1.3 本文主要工作

在当前的研究中，大多数处理器中都配备有向量指令，而一般很少将 AVX 指令应用于 RAID 的计算过程中，更多的是偏重于对 RAID 算法的优化和创新。本研究将分别针对 RAID5 和 RAID6 两种阵列实施加速，RAID5 的编码计算仅仅为奇偶校验，RAID6 在 RAID5 的基础上再引入 Reed-Solomon 编码。本研究将实现 RAID5 和 RAID6 编解码，并使用 AVX 指令分别为两种编码实施加速，与通过 CPU 实施编码计算的方案进行对比。

1.4 论文组织

第一章：引言。首先介绍了本文的选题背景、国内外研究现状、研究目的。介绍目前 RAID 编码和 AVX 指令研究现状，最后简单介绍本文的主要工作和本文的组织结构。

第二章：背景知识简介。介绍本文所研究的基于 AVX 指令的 RAID 编码加速用到相关技术知识。包括 RAID 的各个级别的一些技术细节，Reed-Solomon 编码原理和思路，伽罗华域，RAID6 编码与解码的过程和原理。同时对 AVX 指令集进行简单介绍。

第三章：基于 AVX 指令的 RAID 编解码加速技术。对于本文提出的算法，本章介绍了将 AVX 指令用于 RAID 中改进的算法实现。通过使用 AVX 指令替换 RAID 编解码中的一些步骤，实现加速的效果。重点讲述如何将 AVX 指令用于 RAID 编解码。

第四章：实验结果与分析。通过使用 AVX 指令进行 RAID 编解码与使用 CPU 实施编解码的方案进行对比，验证基于 AVX 指令的 RAID 编解码的有效性和正确性。实验结果表明，使用 AVX 指令优化后可以提升编解码速度，确保正确率的同时，在计算速度方面有着巨大的提升。同时分析其加速原因和瓶颈制约。

第五章：总结与展望。总结本文的工作并对未来工作进行展望。

第二章　背景知识简介

本章首先对 RAID 系统进行简单介绍，同时介绍了 RAID 系统的分类；然后详细介绍了 Reed-Solomon 算法的实现思路，对其实现过程和原理进行详细介绍；随后介绍了 RAID6 的编解码实现原理；最后对 AVX 指令进行简单介绍。

2.1　RAID 编码介绍

RAID 编码即冗余磁盘阵列技术 [10, 11, 12]，该技术使用很多独立的磁盘，将其按照一定的规则进行排列和布局，从而组成容量巨大的磁盘组。RAID 技术在磁盘组的基础上利用条带化技术，在多个磁盘上实现数据成块并发存取，从而减少磁盘寻道时间，加快存取速度，提升存储系统的吞吐量。同时利用冗余技术，添加数据的冗余存储，提供单块磁盘无法具备的容错功能，确保存储系统的可靠性。RAID 系统主要包括 RAID0～RAID6 共 7 个级别，各种 RAID 级别概述如下。

(1) RAID0: 是最早出现的 RAID 模式，利用条带化技术，将原始数据切分为多个块，存储到不同磁盘中。在存取数据时，可以通过并行的方式同时访问多块磁盘，从而提升数据存取速度。其最大的优点是可以成倍提升磁盘组的容量，同时与单块磁盘的存取速度相同，但由于使用的磁盘数量多，并且没有利用冗余技术，导致磁盘可靠性较低。

(2) RAIDl: 采用镜像技术，将磁盘的数据存储到镜像磁盘中，能在不影响性能的情况下最大提升存储系统的可靠性。但由于镜像磁盘的存在，导致其存储效率只有 50%，会显著增加成本，因此只适用于存储空间充足且可靠性要求高的场合。

(3) RAID2: 利用条带化技术，同时添加了海明码校验。对原始数据按照位或字节进行分割，通过海明码计算出冗余数据，将原始数据和冗余数据共同存储在存储系统中。由于添加了冗余存储，具有一定的数据查错和纠错能力。

(4) RAID3: 利用条带化技术，同时添加了奇偶校验。对原始数据按照位或字节进行分割，通过奇偶校验计算出冗余数据，将原始数据和冗余数据共同存储在存储系统中。相比于 RAID2，RAID3 具有更高的存储效率，但采用了奇偶校验的方法，因此 RAID3 只具有差错功能，不具有纠错功能。

(5) RAID4: 是对 RAID3 的改进方法。在条带化的过程中按照块对原始数据进行分割，因此在存取数据时能够减少磁盘寻道时间。但由于所有校验结果都存储在校验磁

盘中，对任意数据的存取都会涉及校验磁盘的操作，导致了 I/O 瓶颈。

(6) RAID5: 是对 RAID4 的改进方法，采用了分布式存储结构。RAID5 将校验结果均匀分布在所有磁盘中，从而避免 RAID4 中存在的 I/O 瓶颈问题。相比于其他 RAID 结构，RIAD5 读写性能高、空间利用率高、数据可靠性强，不具有瓶颈问题，是一种比较均衡的技术方案，得到了工业界的广泛的应用。

(7)RAID6: 是对 RAID5 的进一步优化，采用了 P 和 Q 两个冗余磁盘，能够容忍任意两个磁盘并发故障，具有更高的可靠性。RAID6 的实现方式有很多种，例如 Reed-Solomon 码、Cauchy Reed-Solomon 码、EVENODD 码、RDP 码、Liberation 码、Liber8tion 码、Blaum-Roth 码、X-Code 码、B-Code 码、P-Code 码等。其缺点在于计算的过程复杂，存取的效率较低。

目前，RAID5 和 RAID6 是工业界中常用的两种级别，本文也主要针对这两种 RAID 级别的加速进行研究。

2.2 Reed-Solomon 算法

2.2.1 Reed-Solomon 算法概述

Reed-Solomon（RS）码是一种支持任意码长、容错度的纠删码，通过有限域 $GF(2^w)$ 构造，编码和解码过程中的计算都是有限域运算。对于给定的数据磁盘个数 n 和冗余磁盘个数 m，参数 w 满足 $2^w > m+n$ 时，可以容忍任意 m 个磁盘并发故障。RS 码的实现方式有两种，分别为基于范德蒙德矩阵的 RS 码 [2] 和基于柯西矩阵的 RS 码 [3]。基于范德蒙德矩阵的 RS 码通过构造正对数表和反对数表，加快编码和解码运算。基于柯西矩阵的 RS 码可以将原本复杂的运算转变为 XOR 运算，从而降低编码和解码复杂度，但其复杂度仍高于理论下界，因此文献 [5] 研究了如何进一步优化基于柯西矩阵的 RS 码。

当数据磁盘个数为 n、冗余磁盘个数为 m 时，基于范德蒙德矩阵的 RS 码用 RS$(n+m, m)$ 表示，冗余数据为原始数据和生成矩阵的乘积，生成矩阵由 $n \times n$ 的单位矩阵和 $n \times m$ 的冗余矩阵组成。图2-1展示了 RS$(8, 5)$ 编码的计算过程。

在实际应用中，利用 RS 码编码的磁盘阵列如图2-2所示。其中包括 n 个数据磁盘和 m 个冗余磁盘，可以容忍任意 m 个磁盘并发故障。在实际应用中，RS 码编码和解码都需要复杂的运算，并且每一次数据的存取都会涉及冗余盘的读取，包括计算冗余数据和验证冗余数据正确性，这将导致高额的计算和磁盘 I/O。

本文采用基于范德蒙德矩阵的 RS 码，其实现过程主要有三个步骤：使用范德蒙德

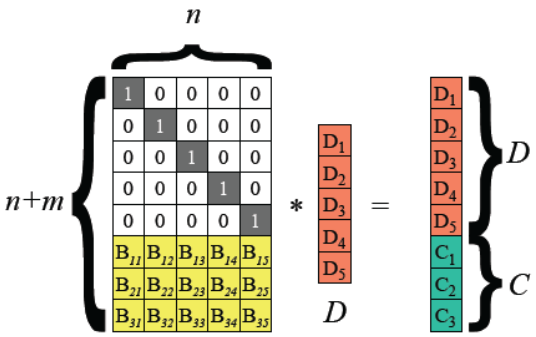

图 2-1 RS(8,5) 编码计算规则

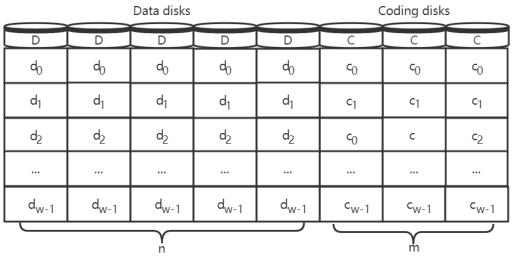

图 2-2 RS 编码的磁盘阵列

矩阵计算和维护校验和，使用高斯消元法从故障中恢复，使用 GF 加速运算[4]。下面将具体说明 RS 算法的实现过程。

2.2.2 计算和维护校验和

RS 算法是以范德蒙德矩阵为基础，对于数据 $M[D_1, D_2, D_3, D_4, D_5](n = 5, m = 2)$，$C_1$、$C_2$ 两个校验块的数据将由 F_1，F_2 两个函数得到。

$$C_1 = F_1(D_1, D_2, D_3, D_4, D_5),$$

$$C_2 = F_2(D_1, D_2, D_3, D_4, D_5),$$

对于具有 n 个数据块的数据，采用 m 个校验块进行编码。我们把每个数据块分成字处理，字的长度为 w bits，w 可以由程序设计者自行选择，因此每个数据块包含 $l = (k\ bytes)(\frac{8\ bits}{byte})(\frac{1\ word}{wbits}) = \frac{8k}{w}$ 个字，编码函数按字逐个操作。为了简化描述，假设每个数据块仅包含 1 个字，问题简化为 n 个数据字 $d_1, d_2, \ldots, d_n$ 和通过计算得到的 m 个检验和字 $c_1, c_2, \ldots, c_n$。定义 F_i 为数据块的线性运算，则：

$$c_i = F_i(d_1, d_2, \ldots, d_n) = \sum_{j=1}^{n} d_j f_{i,j},$$

如果把数据和检验和看成向量 D 和 C，函数 F_i 是矩阵 F 的行向量，则原等式可用如下等式表示：

$$FD = C,$$

定义 F 为 $m \times n$ 的范德蒙德矩阵：$f_{i,j} = j^{i-1}$，则上述等式可以转变成：

$$\begin{bmatrix} f_{1,1} & f_{1,2} & \dots & f_{1,n} \\ f_{2,1} & f_{2,2} & \dots & f_{2,n} \\ \vdots & \vdots & & \vdots \\ f_{m,1} & f_{m,2} & \dots & f_{2,n} \end{bmatrix} \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_n \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & 2 & 3 & \dots & n \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & 2^{m-1} & 3^{m-1} & \dots & n^{m-1} \end{bmatrix} \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_n \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix},$$

当数据字 d_j 变为 d'_j 时，每个校验和字都要跟着改变。这可以通过减去 d_j 有关部分在加 d'_j 有关部分得到，使用 $G_{i,j}$ 函数表示如下：

$$c'_i = G_{i,j}(d_j, d'_j, c_i) = c_i + f_{i,j}(d'_j - d_j).$$

2.2.3 从故障中恢复

为了解释从故障中恢复的过程，定义矩阵 A 和向量 E 为，$A = \begin{bmatrix} I \\ E \end{bmatrix}$，$E = \begin{bmatrix} D \\ C \end{bmatrix}$，于是就可以得到等式 $AD = E$:

$$\begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & \dots & 1 \\ 1 & 1 & 1 & \dots & 1 \\ 1 & 2 & 3 & \dots & n \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & 2^{m-1} & 3^{m-1} & \dots & n^{m-1} \end{bmatrix} \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_n \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_n \\ c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix},$$

我们可以把系统中的每个设备看作是矩阵 A 和向量 E 的对应行。当设备发生故障时，我们通过从矩阵 A 和向量 E 中删除设备的对应行来反映故障。结果得到一个新的矩阵 A' 和一个新的向量 E'，他们符合下面的方程：

$$A'D = E',$$

假设有 m 个设备故障，则 A' 为 $n \times n$ 矩阵。因为矩阵 F 定义为范德蒙德矩阵，矩阵 A 的任意 n 行都是线性无关的，因此 A' 为非奇异矩阵，D 的值可以通过矩阵 $A'D = E'$

经过高斯消元法得到恢复。

一旦 D 的值得到，任何失效的 C_i 都可以重新计算。以此类推，当有 m 个设备故障时，可以从 A' 中任选 n 行通过高斯消元法进行数据恢复。因此，该系统最多可以容忍 m 个设备同时失效。

2.2.4 迦罗华域（GF）运算

RS 算法的一个主要问题是，计算的域和范围是固定长度为 w 的二进制字。虽然上面的代数运算在所有元素都是无限精度实数时保证是正确的，但是我们必须确保它对于这些固定大小为 w 的字是正确的。处理这些运算时的一个常见错误是对整数进行模运算，但因为没有为所有的元素定义除法，使得高斯消元在很多情况下是不可解的。因此，我们必须使用具有多于 $m+n$ 个元素的域执行加法和乘法。

另一个问题是，RS 算法的增加了 I/O 过程中的计算量，必将影响到 I/O 的整体性能，因此，RS 算法实现过程中一个很重要的问题就是如何高效计算 PQ 校验码？对于 P 编码，计算量比较小，直接可以进行 XOR 运算。但是，Q 编码涉及到大量的乘加运算，没有特殊的硬件支撑，很难提高 I/O 的性能。对于乘法运算，我们可以通过对数运算降低计算复杂度，把复杂的乘法运算转换成了加法运算（XOR）。

$$F \times D = C \leftrightarrow \log_g F + \log_g D = \log_g C,$$

$$C = g^{\log_g F + \log_g D},$$

针对这些问题，在这里引入迦罗华域（GF），迦罗华域是一个有限循环域，域中的元素通过本原多项式产生。RS 算法采用 $GF(2^8)$，本原多项式为：$P(x) = x^8+x^4+x^3+x^2+1$。通过这个本原多项式，可以得到有限域中的所有元，如表2.1所示。考虑到迦罗华域的有限性，我们可以把域中元素对应的对数/逆对数做成表格，通过查表的方式，实现 Q 编码的快速运算。

仔细观察迦罗华域元素生成表格，我们可以发现，在迦罗华域中进行对数/逆对数的运算很简单，因为所有域元素都是本原元 a 的 n 次幂，所以，可以很容易的得到对数/逆对数表格。由此，我们可以得出，迦罗华域是 RS 算法的运算域，Q 码的运算可以通过对数/逆对数表格转换成普通的加法运算，在迦罗华域加法运算等价于 XOR。因此，RS 算法所有运算都可以转换成 XOR 异或运算。

表 2.1 $GF(2^8)$ 上元素及其二进制代码

$GF(2^8)$ 域元素	多项式表示	二进制代码
0	0	0000 0000
α^0	1	0000 0001
α^1	α^1	0000 0010
α^2	α^2	0000 0100
α^3	α^3	0000 1000
α^4	α^4	0001 0000
α^5	α^5	0010 0000
α^6	α^6	0100 0000
α^7	α^7	1000 0000
α^8	$\alpha^4+\alpha^3+\alpha^2+1$	0001 1101
α^9	$\alpha^5+\alpha^4+\alpha^3+\alpha^1$	0011 1010
...	...	...
α^{254}	$\alpha^7+\alpha^3+\alpha^2+\alpha^1$	1000 1110

2.2.5 Reed-Solomon 算法总结

经过上述讨论，将 RS 算法总结如下：

（1）选择一个 w 值，满足 $2^w > m+n$，通常选择 $w=8$ 或 $w=16$，这样字可以选取在字节的边界。

（2）根据选择的 w 值，构造对数表和逆对数表。

（3）建立 $m \times n$ 的范德蒙德矩阵 F，其中 $f_{i,j} = j^{i-1}(1 \le i \le m, 1 \le j \le n)$，并且乘法运算是在 $GF(2^w)$ 进行的。

（4）使用矩阵 F 和数据块计算和维护校验和，所有运算是在 $GF(2^w)$ 进行的。

（5）当至多 m 个设备出现故障，其恢复方法为：选择任意 n 个剩余的设备，并构建前面定义的矩阵 A' 和向量 E'，通过 $A'D = E'$ 求解 D。这样就可以恢复数据设备。数据设备恢复后，使用矩阵 F 重新计算失效的校验和。

2.3 RAID6 编码与解码

RAID6 的编解码有多种实现方式，本文使用基于范德蒙德矩阵的 RS 码实现 RAID6。

2.3.1 RAID6 中 P 校验、Q 校验的生成

RAID6 通过两块数据冗余盘实现条带数据的保护，最坏情况下可以允许两块磁盘的同时损坏。为了达到这个目的，RAID6 有两种校验码，P 码和 Q 码。从数学的角度讲，P、Q 校验码的计算方法如下：

$$P = D_1 \oplus D_2 \oplus \cdots \oplus D_n,$$

$$Q = K_1 \times D_1 \oplus K_2 \times D_2 \oplus \cdots \oplus K_n \times D_n,$$

在使用 $GF(2^8)$ 的性质后，令数据块的大小 $w = 8$，令系数 $K_1, K_2, \ldots, K_n \in GF(2^8)$，构造对数表 $gflog[\,]$ 和逆对数表 $gfilog[\,]$，则可以定义伽罗瓦域乘和伽罗瓦域除的规则如图2-3：

```
#define NW (1 << w)
int mult(int a, int b) {
  int sum_log;
  if (a == 0 || b == 0) return 0;
  sum_log = gflog[a] + gflog[b];
  if (sum_log >= NW-1) sum_log -= NW-1;
  return gfilog[sum_log];
}
int div(int a, int b) {
  int diff_log;
  if (a == 0) return 0;
  if (b == 0) return -1;
  diff_log = gflog[a] - gflog[b];
  if (diff_log < 0) diff_log += NW-1;
  return gfilog[diff_log];
}
```

图 2-3 伽罗瓦域乘和伽罗瓦域除的 C 语言代码

运用 GF 运算后，Q 校验码的计算方法改进为：

$$Q = mult(K_1, D_1) \oplus mult(K_2, D_2) \oplus \cdots \oplus mult(K_n, D_n).$$

2.3.2 RAID6 数据恢复

上述算法可以确保在任意两块磁盘损坏的情况下都可恢复原数据，例如在六块磁盘的情况下 $(m = 4, n = 2)$，若损失两块磁盘数据，那么将会出现以下 4 类情况（下述公式中 × 代表伽罗瓦域乘，/ 代表伽罗瓦域除）：

（1）丢失 P、Q，剩余 D_1、D_2、D_3、D_4

这是最简单的情况，数据并没有丢失，重新计算校验和即可。

（2）丢失 P、D_1，剩余 Q、D_2、D_3、D_4

根据 Q 的计算公式可以推出 D_1 的计算公式如下，D_1 恢复后，重新计算 P 校验和。

$$D_1 = (D_1{\times}K_2 \oplus D_3{\times}K_3 \oplus D_4{\times}K_4 \oplus Q)/K_1,$$

（3）丢失 Q、D_1，剩余 P、D_2、D_3、D_4

根据 P 的计算公式可以推出 D_1 的计算公式如下，D_1 恢复后，重新计算 Q 校验和。

$$D_1 = P \oplus D_2 \oplus D_3 \oplus D_4,$$

（4）丢失 D_1、D_2，剩余 P、Q、D_3、D_4

这是最难处理的情况，可以根据二元一次方式的标准解法进行求解：

$$\begin{aligned}
&P = D_1 \oplus D_2 \oplus D_3 \oplus D_4,\\
&Q = K_1 \times D_1 \oplus K_2 \times D_2 \oplus K_3 \times D_3 \oplus K_4 \times D_4,\\
\Longrightarrow &D_1 = P \oplus D_2 \oplus D_3 \oplus D_4,\\
\Longrightarrow &Q = K_1 \times (P \oplus D_2 \oplus D_3 \oplus D_4) \oplus K_2 \times D_2 \oplus K_3 \times D_3 \oplus K_4 \times D_4,\\
\Longrightarrow &D_2 = (P \times K_1 \oplus D_3 \times K_1 \oplus D_4 \times K_1 \oplus D_3 \times K_3 \oplus D_4 \times K_4 \oplus Q)/(K_1 \oplus K_2),\\
\Longrightarrow &D_1 = P \oplus D_2 \oplus D_3 \oplus D_4.
\end{aligned}$$

2.4 AVX 指令简介

指令是计算机程序发给计算机处理器的命令，指令通常有标量版本和矢量版本，如图2-4[13] 所示。SIMD（Single Instruction Multiple Data，单指令多数据）模式是指在一个 CPU 指令执行周期内，一个指令可以对多个数据进行操作，实现多数据并行处理，极大的提高了数据处理的效率。矢量版本通过在 SIMD 模式下并行处理寄存器中的数据来操作，标量版本只对每个寄存器中的一个条目进行操作。这种区别可以减少某些算法的数据移动，从而提供更好的总体吞吐量。

在 1996 年，英特尔提出的 MMX（Multi Media extensions）多媒体扩展指令集突破了 SIMD 指令集规则。SSE 系列的指令在 1999 年将其对于矢量处理的能力由之前的 64 位提升到了 128 位，AVX 则将 128 位的寄存器扩展到 256 位，最新的 AVX-512 将 256 位的寄存器扩展到 512 位。AVX 为了加强浮点数的运行能力，加入了大量的浮点运算命令，提高了 3D 性能，支持更多复杂的图像处理显示，成本降低，体积更小，使得 GPU 的优势不复存在[14]。

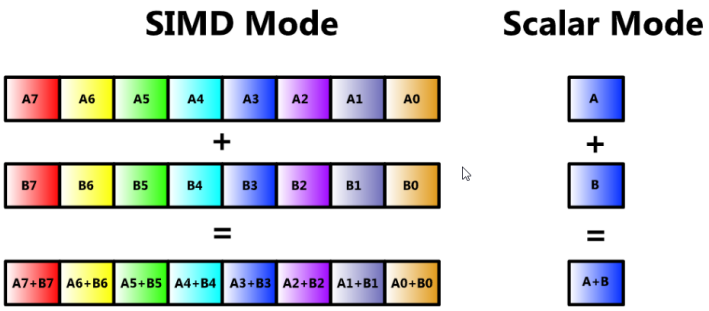

图 2-4 SIMD 与标量操作

2008 年英特尔公布了 AVX 指令集的规范，与此同时，在 Sandy Bridge 的架构上准备推出一套全新的指令集 AVX。AVX，英文全称 Advanced Vector Extensions，即高级矢量扩展集合。它是对 SSE 指令集的一个扩展，新增加了 8 个 256 位的寄存器，为 YMM0-YMM7。同时将 128 位寄存器 XMM SSE 指令集扩展到了 256 位寄存器 YMM AVX 指令集，即寄存器 YMM 低位的 128 位依然使用 SSE 中的 XMM 寄存器，使得 AVX 指令集对 SSE 系列指令集有很好的兼容性，所有的 SSE/SSE2/SSE3/SSSE3/SSE4 指令都是能够被 AVX 全面兼容的。针对密集型浮点运算，AVX 进行矩阵计算的时候将比 SSE 技术快 90%，并且新增加的指令在处理性能上比 SSE 系列更为先进。128 位的 XMM 寄存器扩展到 256 位的 YMM 寄存器示意图如图2-5[13] 所示。

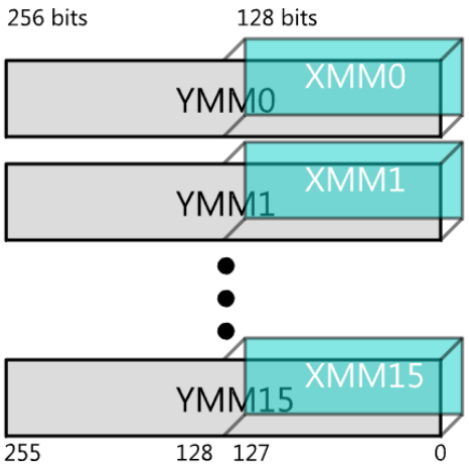

图 2-5 XMM 寄存器与 YMM 寄存器扩展示意图

第三章　基于 AVX 指令的 RAID 编解码加速技术

为了加快 RAID 编解码速度，本章提出基于 AVX 指令的 RAID 编解码加速技术，并使用该技术实现 RAID5 和 RAID6 编解码加速。本章首先介绍基于 AVX 指令加速 RAID5 编解码；随后详细介绍基于 AVX 指令加速 RAID6 编码，包括整体思路和对其中难点的实现思路；最后介绍基于 AVX 指令加速 RAID6 解码。

3.1　基于 AVX 指令加速 RAID5 编解码

RAID5 采用块交叉存取的分布式校验磁盘阵列，编码和解码的主要的计算部分都是对数据采用奇偶校验法，通过 XOR 生成校验数据。因此使用 AVX 指令对 RAID5 进行编解码加速的主要方面就是加快奇偶校验的计算速度。

在本文 RAID5 的实现过程中，对文件进行读取的过程采用以字节为单位读取，在计算的过程中也是以字节（8 位）为单位进行 XOR。为了充分利用 AVX 指令进行加速，使用 AVX 指令中__m256i 数据类型，可以同时将 32 个字节的数据转变为__m256i 类型，从而实现同时计算 256 位的 XOR 运算。

基于 AVX 指令加速 RAID5 编解码具体过程如下：

1) 读取原始数据。
2) 以 256 位为单位，将原始数据分割转换为__m256i 类型。
3) 使用__m256i 类型进行异或运算。
4) 将__m256i 类型转换回以 8 位为单位。
5) 将运算结果保存在文件中。

AVX 过程使用到的函数如下：

1) _mm256_setzero_si256：对__m256i 类型数据进行初始化，赋值 0。
2) _mm256_lddqu_si256：将 256 位整型数据从未对齐内存加载到__m256i 类型中。在本文中，使用的是 32 个连续的 8 位字符，可以看成 256 位整型数据，从而转变为__m256i 类型。
3) _mm256 _xor _si256：对 __m256i 数据进行 XOR 运算。
4) _mm256_storeu_si256：将 256 位的整数数据存储到内存中。

基于 AVX 指令加速后的 RAID5 编解码，减少了代码中 XOR 的计算次数和 CPU 指令执行的次数，数据的运算效率得到明显的提升。假设数据盘大小 k 字节，共包含 n

个数据盘，理论上算法复杂度为 $O(nk/32)$。但在实际的实验和测试中，由于要初始化 AVX 指令，同时数据的读取、转化和保存需要耗时，因此算法的总体的时间复杂度降为 $O(nk/4.2)$。在使用不同数据规模测试时，发现改进后算法性能十分稳定，数据规模的变化不会对时间复杂度造成影响，时间复杂度稳定在 $O(nk/4.2)$ 左右。

3.2 基于 AVX 指令加速 RAID6 编码

3.2.1 算法设计思路

RAID6 采用 P 和 Q 两个冗余磁盘，能够容忍任意两个磁盘并发故障。P 校验和的主要计算部分是对数据采用奇偶校验法，通过 XOR 生成。Q 校验和的主要计算部分是伽罗瓦域乘 (图2-3) 计算，伽罗瓦域乘又可以细分为格式转换、条件语句判断、取模运算、从数组中取数、加法等步骤。因此，主要使用 AVX 指令加快上述步骤。

在本文 RAID6 编码的实现过程中，对文件进行读取的过程采用以字节为单位读取，在计算的过程中也是以字节（8 位）为单位进行 XOR 和伽罗瓦域乘。为了充分利用 AVX 指令进行加速，使用 AVX 指令中__m256i 数据类型，可以同时将 32 个字节的数据转变为__m256i 类型，从而实现同时计算 256 位。原本计算伽罗瓦域乘是在域 $GF(2^8)$ 上进行计算，以 8 位为计算单位，在利用 AVX 指令进行加速时，由于同时计算 256 位，可以考虑在域 $GF(2^{16})$ 上进行计算，即以 16 位为计算单位。

基于 AVX 指令加速 RAID6 编码具体过程如下：

1) 读取原始数据。
2) 以 256 位为单位，将原始数据分割转换为__m256i 类型。
3) 使用__m256i 类型进行异或运算。
4) 使用__m256i 类型进行伽罗瓦域乘运算。
5) 将__m256i 类型转换回以 8 位为单位。
6) 将运算结果保存在文件中。

AVX 过程使用到的函数如下：

1) _mm256_setzero_si256：对__m256i 类型数据进行初始化，赋值 0。
2) _mm256_set_epi8：使用 32 个 8 位整数初始化__m256i。
3) _mm256_set1_epi32：使用 1 个 32 位整数初始化__m256i 中 8 个 32 位整数。
4) _mm256_lddqu_si256：将 256 位整型数据从未对齐内存加载到__m256i 类型中。
5) _mm256_i32gather_epi32：使用 32 位索引从内存中加载 32 位整数，存储到__m256i 中。用于从 int 数组中读取数据存储到__m256i 中。

6) _mm256_storeu_si256：将 256 位的整数数据存储到内存中。
7) _mm256 _xor _si256：对 _ _m256i 数据进行异或运算。
8) _mm256_and_si256：对 _ _m256i 数据进行与运算。
9) _mm256_or_si256：对 _ _m256i 数据进行或运算。
10) _mm256_andnot_si256：对 _ _m256i 数据进行先取非，在进行与运算。
11) _mm256_add_epi32：对 _ _m256i 变量中 8 个 32 位数据进行加法运算。
12) _mm256_shuffle_epi8：根据输入的位置参数将_ _m256i 中的 32 个 8 位整数改变存储顺序。
13) _mm256_extractf128_si256：根据输入的参数提取_ _m256i 中的前 128 位或后 128 位。
14) _mm256_cvtepu8_epi16：通过添加前导 0，将_m128i 中 16 个 8 位整数扩展为 16 个 16 位整数，存储在_ _m256i 中。
15) _mm256_insertf128_si256：根据输入参数，在_ _m256i 的指定位置插入_ _m128i。
16) _mm256_cmpgt_epi32：用于比较两个_ _m256i 类型中的 8 个 32 位整数是否满足大于关系，若对应位置的 32 位整数满足大于关系，则结果中对应位置 32 位为全 1，否则为全 0。
17) _mm256 _cmpeq _epi32：用于比较两个_ _m256i 类型中的 8 个 32 位整数是否相等，若对应位置的 32 位整数相等，则结果中对应位置 32 位为全 1，否则为全 0。
18) _mm256_slli_epi32：根据输入参数，将_ _m256i 类型中 8 个 32 位整数左移。

使用 AVX 指令实现伽罗瓦域乘可以细分为格式转换、条件语句判断、取模运算、从数组中取数、加法等步骤，下面将对前三个步骤的实现方法进行详细的描述：

3.2.2 格式转换

在进行伽罗瓦域乘的运算过程中，包含多个从数组取数的步骤，如 $gflog[a]$，受 AVX 指令的限制，使用 AVX 指令从数组中非连续取数只能使用_mm256_i32gather_epi32 函数，因此需要对存储 32 个 8 位整形的_ _m256i 类型数据进行格式转换，将其转换为存储 8 个 32 位整形的_ _m256i 类型数据。

方法一：为实现这一要求，可以利用_mm256_extractf128_si256 函数将_ _m256i 类型数据分割为两个_ _m128i 类型数据，在利用_mm256_cvtepu8_epi16 函数将_ _m128i 类型数据扩展为_ _m256i 类型数据，将次过程重复两遍即完成了数据格式转换。如图3-1 。

上述是一种简单直观的方法，实现了从 8 位整数到 32 位整数的转换，但为从 32 位整数到 8 位整数的转换制造了困难。用这种方式进行格式转换后，原始数据按顺序分

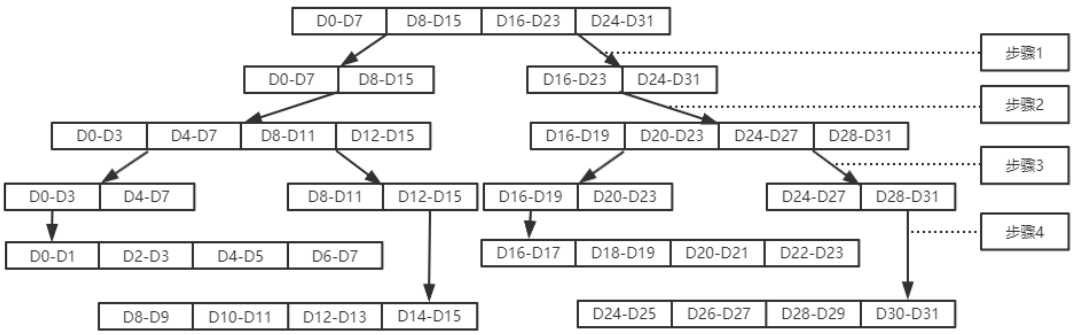

图 3-1 格式转换

布在 4 个__m256i 类型数据中，因此只能将数据从 4 个__m256i 类型数据中读出，采用_mm256_set_epi8 函数转换回原格式。但这种转换方式增加了多余的 I/O 操作，如：从__m256i 类型中读出 8 个整形，将 32 个整形写入__m256i 类型，转换过程中的数据存储操作等，导致极大的时间浪费。为了提升数据格式转换的效率，我们对方法一进行了优化，得到方法二。

方法二：为了提升数据格式转换的效率，我们采用了改变原始数据顺序的方法图3-2。为了作图简单，我们使用 8 个数据讲述方法二的思路。通过改变原始数据的顺序，使得通过方法一格式转换后，原始数据交叉存储在 4 个__m256i 类型中，从而可以通过移位运算和或运算恢复原始数据。

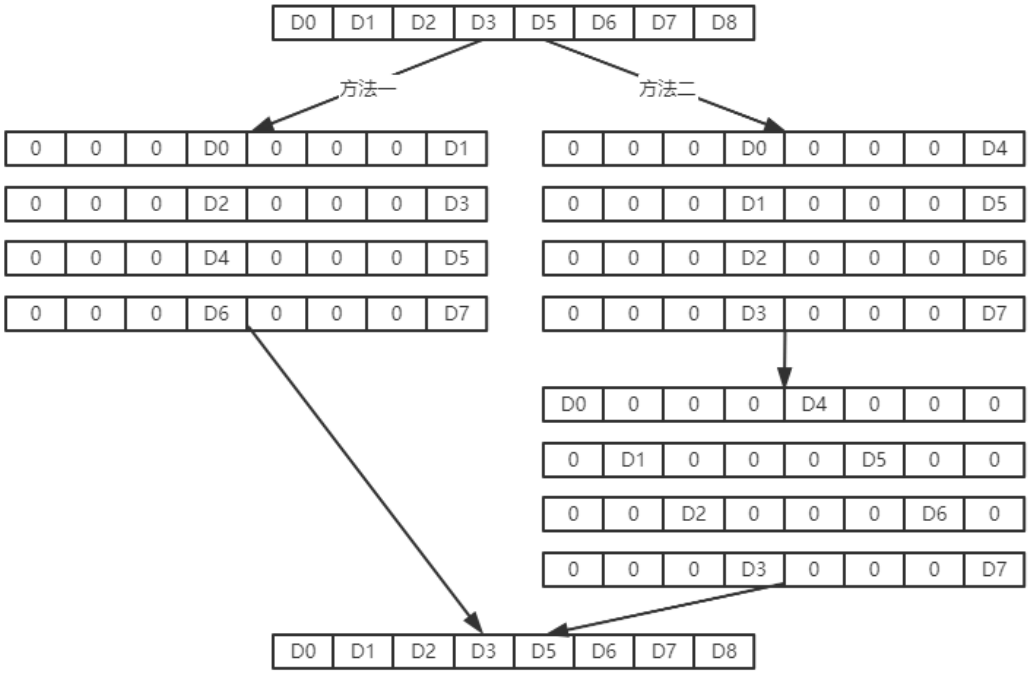

图 3-2 顺序存储和交叉存储

利用_mm256_shuffle_epi8 函数，改变原始数据存储顺序，在通过方法一进行运算，

即可得到交叉存储的格式转换结果。将数据恢复原位时，只需通过_mm256_slli_epi32函数，将 4 个__m256i 分别左移 24 位、16 位、8 位、0 位，在利用或运算即可恢复。

方法二相比于方法一，在格式转换过程中，增加了一次重排序、3 次移位运算、3 次或运算，减少恢复过程中的数据读取和存储时间。增加的计算过程是寄存器间运算，减少的过程是寄存器和内存间数据转换，因此极大的加快了运行的效率。通过实际测试，方法二的运算时间为方法一的 50%，与理论分析结果相同。

3.2.3 条件语句判断

在进行伽罗瓦域乘的运算过程中，包含两个条件判断，受 AVX 指令的限制，使用 AVX 指令没有执行条件语句的指令，因此只能通过其他方式实现条件语句判断。

两个条件语句分别是判断 a（数据）和 b（系数）是否为 0，若是 0 则直接返回 0，通过 AVX 指令加速时，将 32 个不同的 a 合并为一个__m256i 类型，b 对应同一组数据是相同的，因此 b 的条件语句可以直接通过原方法判断，a 则相对复杂。

对于 a 的条件判断，采用图3-3 的方法，为了作图简单，我们使用 4 个数据讲述条件语句判断思路。对于输入数据 $D = [D0, D1, D2, D3]$，经过正常计算后得到数据 $D' = [D0', D1', D2', D3']$，为了实现条件语句判断，我们将 D 与全 0 数据进行比较，若数据 D 中 D0 为 0，则经过比较后得到的数据 P 在 D0 位置为全 1，若数据 D 中 D1 非 0，则经过比较后得到的数据 P 在 D1 位置为全 0。将 D 运算后得到的 D’ 与 ~P 进行与运算，即可实现条件语句判断的效果。

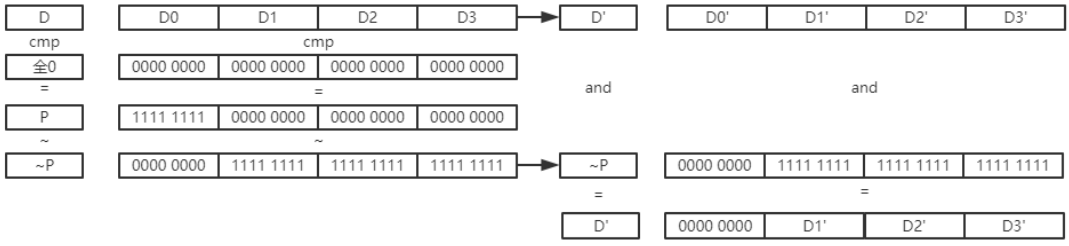

图 3-3 条件语句判断

在条件语句的判断过程中，增加了多余的操作，例如：比较、与、取非，同时增加了额外的运算，在非 AVX 指令运算的过程中，判断 $a = 0$ 可以直接返回 0，而通过 AVX 指令加速后，这些值为 0 的数据也和非 0 数据一样进行了计算，因此增加了额外运算。尽管增加了一些额外操作，相比于 AVX 指令提供的并行运算，利用 AVX 指令整体上仍加快了运算速度。

3.2.4 取模运算

在进行伽罗瓦域乘的运算过程中，为了使计算结果仍落在伽罗瓦域中，包含了判断计算结果是否越界，若计算结果越界，则减去域的大小，使结果仍落在域中。在实现这条规则时，使用了取模运算。受 AVX 指令的限制，使用 AVX 指令没有执行取模运算的指令，因此只能通过其他方式实现取模。

在使用域 $GF(2^8)$ 时，域大小为 256，因此需要对计算结果进行%255 运算。对于小于 255 的数，直接取二进制的后 8 位就得到结果，对于大于等于 255 的数，需要先将原数加 1，再取二进制后 8 位得到结果。

在 AVX 指令下实现取模，需要使用_mm256_cmpgt_epi32 函数实现 255 与原数的比较，将比较结果加 1 后与原数相加，最后将相加结果和 255 求与运算，即可得到所需结果。

3.2.5 算法总结

基于 AVX 指令加速后的 RAID6 编码，减少了代码中 XOR 的计算次数、函数调用次数、CPU 指令执行的次数，数据的运算效率得到明显的提升。假设数据盘大小 k 字节，共包含 n 个数据盘，理论上算法复杂度为 $O(nk/32)$。但在实际的实验和测试中，由于要初始化 AVX 指令、数据的转化和保存需要耗时、基于 AVX 指令实现过程中添加了一些多于步骤，因此算法的总体的时间复杂度降为 $O(nk/3.4)$。在使用不同数据规模测试时，发现改进后算法性能十分稳定，数据规模的变化不会对时间复杂度造成影响，时间复杂度稳定在 $O(nk/3.4)$ 左右。

当使用域 $GF(2^{16})$ 时，可以使代码中计算伽罗瓦域乘的数目减少到原本的一半，理论上算法复杂度为 $O(nk/64)$。但在实际的实验和测试中，由于对数表和逆对数表的增大，导致表不能完全存放在一级缓存中，计算速度与使用域 $GF(2^8)$ 近似，时间复杂度稳定在 $O(nk/3.3)$ 左右。在使用不同数据规模测试时，发现改进后算法性能十分稳定，数据规模的变化不会对时间复杂度造成影响，时间复杂度稳定在 $O(nk/3.3)$ 左右。

3.3 基于 AVX 指令加速 RAID6 解码

RAID6 的解码过程，按照2.3.2节所述，在 RAID6 编码的基础上增加了伽罗瓦域除(图2-3) 计算。根据对伽罗瓦域除实现过程的分析，和伽罗瓦域乘思路几乎相同。区别在于伽罗瓦域乘中为了使计算结果仍落在伽罗瓦域中，使用了取模运算。在伽罗瓦域除实现过程中，为了实现计算结果落在伽罗瓦域中，避免计算结果取负，采用了计算结果

越界，则加域的大小，使结果仍落在域中。为实现这条规则，可以采用对计算结果先加域的大小，在对域的大小取模，从而实现。

其余使用 AVX 指令加速 RAID6 解码过程和加速 RAID6 编码类似，本文不再重述。

第四章　实验结果与分析

为了验证基于 AVX 指令的 RAID 编解码加速效果，本章通过对比实验验证了该算法的性能和正确性。本章首先将基于 AVX 指令加速的 RAID5 和 RAID6 编解码与原始算法对比；随后将基于 AVX 指令的 RAID6 编解码与当前主流 RAID6 算法进行对比。

4.1　基于 RAID5 的对比实验

本文使用 C 语言实现了基于 AVX 指令集的 RAID5 编解码。测试平台采用的 CPU 为 Intel Xeon Gold 6150，8 核处理器，16G 内存，系统为 Centos 7.3.1611。实验使用不同大小的文件进行 RAID5 编码，对 AVX 指令加速的优化效果进行验证。

为了更好的理解和验证算法的有效性和稳定性，在同一实验条件和同一数据集下分别进行 RAID5 编码实验和基于 AVX 指令 RAID5 编码实验。实验结果表明，本文提出的 AVX 指令优化后的 RAID5 编码比原有的 RAID5 编码速度更快，算法改进后时间性能提升到原来的 4.2 倍。为进一步验证算法稳定性，实验采用 4 块数据块，单块数据块大小为 106MB、212MB、424MB 进行 RAID5 编码，用相同大小数据块分别在 RAID5 编码和基于 AVX 加速 RAID5 编码上运行，进行时间消耗对比。对比结果如表4.1所示。

表 4.1 RAID5 编码与 AVX-RAID5 编码时间对比

算法	单块数据块大小		
	106MB	212MB	424MB
RAID5 时间（s）	0.42	0.83	1.66
AVX-RAID5 时间（s）	0.10	0.18	0.37

从表4.1中，我们可以看出，基于 AVX 优化的 RAID5 编码相比与传统 RAID5 编码可以节约大约 77% 的时间消耗。这可以说明基于 AVX 优化的 RAID5 编码是有效的。从表格是数据可以看出，两个算法的计算时间和数据块的大小成线性变化，随着数据块规模的增大，两种方法的运行时间均大致呈线性增加，即两种方法的时间复杂度与数据块大小呈线性相关，体现了两种算法的稳定性。

4.2 基于 RAID6 的对比实验

4.2.1 格式转换两种方法对比实验

本文使用 C 语言实现了基于 AVX 指令集的 RAID6 编解码。测试平台采用与 RIAD5 编码相同实验平台。实验使用不同大小的文件进行 RAID6 编解码，对 AVX 指令加速的优化效果进行验证。

对于章节3.2.2中提出的两种格式转换方法，为了更好的比较两种方法的速度，在同一实验条件和同一数据集下分别使用两种方法进行 RAID6 编码测试。实验结果表明，方法二中的改进对于提升格式转换速度具有明显效果，使用方法二改进后编码时间性能提升到方法一的 1.9 倍。为进一步验证算法稳定性，编码实验采用 4 块数据块，2 个检验块，在域 $GF(2^8)$ 上进行计算，单块数据块大小为 106MB、212MB、424MB，用相同大小数据块分别使用方法一和方法二进行数据格式转换，对比结果如表4.2所示。

表 4.2 格式转换两种方法时间对比

算法	单块数据块大小		
	106MB	212MB	424MB
方法一时间（s）	0.82	1.66	3.33
方法二时间（s）	0.43	0.88	1.81

4.2.2 与 Reed-Solomon 算法对比实验

为了更好的理解和验证算法的有效性和稳定性，在同一实验条件和同一数据集下分别进行 Reed-Solomon 算法编解码实验和基于 AVX 指令 Reed-Solomon 算法编解码实验。实验结果表明，本文提出的 AVX 指令优化后的 Reed-Solomon 算法编解码比原有的 Reed-Solomon 算法编解码速度更快，算法改进后编码时间性能提升到原来的 3.2 倍，解码时间性能提升到原来的 3.3 倍。为进一步验证算法稳定性，编码实验采用 4 块数据块，2 个检验块，在域 $GF(2^8)$ 上进行计算，单块数据块大小为 106MB、212MB、424MB，用相同大小数据块分别在 Reed-Solomon 算法编码和基于 AVX 加速 Reed-Solomon 算法编码上运行，进行时间消耗对比，编码对比结果如表4.3所示。解码实验在编码基础上，当出现两块数据块丢失时，对源数据进行恢复，解码对比结果如表4.4所示。

从表4.3和表4.4中，我们可以看出，基于 AVX 优化的 Reed-Solomon 算法编解码相比与传统 Reed-Solomon 算法编解码可以节约大约 70% 的时间消耗。这可以说明基于 AVX 优化的 Reed-Solomon 算法编解码是有效的。从表格是数据可以看出，两个算法的

表 4.3 Reed-Solomon 算法编码与 AVX-Reed-Solomon 算法编码时间对比

算法	单块数据块大小		
	106MB	212MB	424MB
Reed-Solomon 时间（s）	1.40	2.83	5.66
AVX-Reed-Solomon 时间（s）	0.43	0.88	1.81

表 4.4 Reed-Solomon 算法解码与 AVX-Reed-Solomon 算法解码时间对比

算法	单块数据块大小		
	106MB	212MB	424MB
Reed-Solomon 时间（s）	1.99	4.08	8.1
AVX-Reed-Solomon 时间（s）	0.60	1.22	2.42

计算时间和数据块的大小成线性变化，随着数据块规模的增大，两种方法的运行时间均大致呈线性增加，即两种方法的时间复杂度与数据块大小呈线性相关，体现了两种算法的稳定性。

4.2.3 与其他主流算法对比实验

实验的第三部分就是将本文提出的算法与目前其他 RAID6 编解码主流算法进行对比实验测试。实验数据采用 4 块数据块，2 个检验块，在域 $GF(2^8)$ 上进行计算，单块数据块大小为 106MB、212MB、424MB。选取的对比算法为 jerasure 1.2 库 [15] 中的 Reed-Solomon 算法、Reed-Solomon 优化算法、Cauchy Reed-Solomon 算法、Cauchy Reed-Solomon 优化算法、Liberation 算法、Blaum-Roth 算法、Liber8tion 算法，计算参数为 $k = 4, m = 2, w = 8, packetsize = 1024, buffersize = 16384$（Liberation 算法中 $w = 7$，Blaum-Roth 算法中 $w = 10,$）。表4.5和表4.6分别显示了编码和解码的比较结果。

从表4.5和表4.6可以看出，使用 AVX 指令改进后的 Reed-Solomon 算法编解码速度可优于其他 RAID6 编解码主流算法，这充分体现了本文提出的基于 AVX 指令改进算法的优势，表明基于 AVX 指令可以在确保正确性和稳定性的基础上，极大提升算法的时间性能。

表 4.5 AVX-Reed-Solomon 算法编码时间与其他算法编码时间对比

算法	单块数据块大小		
	106MB	212MB	424MB
AVX-Reed-Solomon 时间（s）	0.43	0.88	1.81
Reed-Solomon 时间（s）	1.50	2.98	6.17
Reed-Solomon 优化时间（s）	0.82	1.71	3.32
Cauchy Reed-Solomon 时间（s）	0.92	1.81	3.64
Cauchy Reed-Solomon 优化时间（s）	0.63	1.26	2.62
Liberation 时间（s）	0.61	1.26	2.47
Blaum-Roth 时间（s）	0.57	1.17	2.27
Liber8tion 时间（s）	0.62	1.18	2.30

表 4.6 AVX-Reed-Solomon 算法解码时间与其他算法解码时间对比

算法	单块数据块大小		
	106MB	212MB	424MB
AVX-Reed-Solomon 时间（s）	0.60	1.22	2.42
Reed-Solomon 时间（s）	1.72	3.39	6.84
Reed-Solomon 优化时间（s）	1.72	3.39	6.90
Cauchy Reed-Solomon 时间（s）	1.21	2.46	4.78
Cauchy Reed-Solomon 优化时间（s）	0.74	1.52	2.99
Liberation 时间（s）	0.66	1.37	2.76
Blaum-Roth 时间（s）	0.72	1.44	2.92
Liber8tion 时间（s）	0.71	1.40	2.81

第五章　总结与展望

5.1　工作总结

本文主要的研究是基于 AVX 指令的 RAID 编码加速，主要基于 RAID5 算法和 Reed-Solomon 算法。实验结果表明，本文提出的算法有效提高了 RAID 编解码的速度，下面就本文提出的基于 AVX 指令的 RAID 编码加速进行总结。

算法设计的前期通过对 RAID 编码的方法进行分析，选取了 RAID5 编解码，和 RAID6 编解码中经典算法 Reed-Solomon 算法，通过对背景知识的学习和对算法的研究，实现了两种算法的复现。

针对访存次数多，计算量大的特点，本文提出并设计和实现了基于 AVX 指令的 RAID5 编解码和基于 AVX 指令的 RAID6 编解码。在计算时使用 AVX 指令集进行并行计算，减少了编解码过程中的访存次数，降低了算法的计算量，提高了算法的计算效率，使数据运算效率得到明显提升。实验结果表明，本文提出的算法节约了原来 RAID5 算法 77% 左右的时间消耗、节约了原来 Reed-Solomon 算法 70% 左右的时间消耗。

本研究中还存在一定的问题，对 Reed-Solomon 算法进行优化的过程中，受实验环境的限制，导致只能使用 AVX2 指令（256 位寄存器），可以使用最新的 AVX-512 指令（512 位寄存器），对计算速度做进一步优化。在数据存储的过程中，AVX 指令提供了_mm256_stream_si256 函数，可以将 256 位数据更快的从寄存器存储到内存中。但该函数要求数据在 32 字节边界上对齐才能使用，在本研究中由于使用了 char 数组，导致没有实现在 32 字节边界上对齐，因此使用了另一个存储函数，降低了一定的效率。

5.2　研究展望

本文对数据存储领域的 RAID 编码进行研究和学习，在深入研究的同时发现了目前已有算法的不足，提出使用 AVX 指令优化的 RAID 编码设计与实现，降低了传统算法的计算时间。在对算法改进的探索中，发现了新技术，AVX 指令集，在学习研究了 AVX 指令集后，提出了基于 AVX 指令集的 RAID 编码加速。AVX 指令的的应用为数据存储中的 RAID 技术提出了新颖的思路，但对算法的改进和研究的深度和广度都还远远不够。还有很多方面可以完善。

（1）在研究 RAID 编码加速的过程中，融合了 AVX 技术，对已有算法进行了改进。但实现 RAID 的方法有很多种，本文中只实现了经典的 Reed-Solomon 算法，对于更多

高效、复杂的算法本文没有涉及。随着算法设计的复杂度提高，AVX 指令能提供的帮助也会有所减少。所以，本文提出的亦当做抛砖引玉，为推动 RAID 编码尽一份力，若能为后来的研究人员起到启发的作用，更是本文意义之所在。

（2）通过并行化对原有算法进行优化将成为数据存储领域的一个重要研究方向，通过对 AVX 指令的使用，可以加快数据压缩、分类、存储等方面的研究，以及更多其他研究领域。

参考文献:

[1] 黄志杰. 容错存储系统中的 MDS 阵列码研究 [D]. [S.l.]: 华中科技大学, 2016.

[2] REED I S, SOLOMON G. Polynomial Codes Over Certain Finite Fields[J]. Journal of the Society for Industrial & Applied Mathematics, 1960, 8(2): 300 – 304.

[3] ROTH R M, LEMPEL A. On MDS codes via Cauchy matrices[J]. IEEE Transactions on Information Theory, 1989, 35(6): P.1314 – 1319.

[4] PLANK J S. A Tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-like Systems[J]. Software Practice & Experience, 1996, 27(9): 995 – 1012.

[5] PLANK J S, XU L. Optimizing Cauchy Reed-Solomon Codes for Fault-Tolerant Network Storage Applications[C] // Fifth IEEE International Symposium on Network Computing and Applications, NCA 2006, 24-26 July 2006, Cambridge, Massachusetts, USA. 2006.

[6] 牟永敏, 刘城霞, 张京生, et al. Reed-Solomon 算法在 RAID6 系统中的应用 [J]. 电子学报, 2007, 035(s2): 90 – 94.

[7] BLAUM M, BRADY J, BRUCK J, et al. EVENODD: an efficient scheme for tolerating double disk failures in RAID architectures[J]. IEEE Transactions on Computers, 1995, 44(2): 192 – 202.

[8] 兰玉龙. 容双盘错 RDP 算法的设计与实现 [D]. [S.l.]: 华中科技大学, 2007.

[9] 沙睿彬. RAID6 中 P-Code 编码研究与实现 [D]. [S.l.]: 华中科技大学, 2012.

[10] 谢平. RAID-6 编码布局及重构优化研究 [D]. [S.l.]: 华中科技大学, 2015.

[11] PATTERSON D A, GIBSON G, KATZ R H. A case for Redundant Arrays of Inexpensive Disks (RAID)[J]. ACM SIGMOD Record, 1988, 17(3).

[12] 王能. 磁盘阵列存储系统的分组编码技术研究 [D]. [S.l.]: 中国科学技术大学, 2016.

[13] LOMONT C. Introduction to intel advanced vector extensions[J]. Intel white paper, 2011, 23.

[14] 杜博雅. DNA 序列比对中基于 AVX 指令集的 BWT 算法研究 [D]. [S.l.]: 东北农业大学, 2017.

[15] PLANK J S. The RAID-6 liberation codes[C] // 6th USENIX Conference on File and Storage Technologies, FAST 2008, February 26-29, 2008, San Jose, CA, USA. 2008.